# Corndel DevOps Engineering Programme

**in association with Softwire**

**Module 1:** Project Exercise

Corndel Digital.

DBAR

# Project Exercises

The technical concepts that we'll cover during this course can sometimes feel quite abstract when reading about them or following examples. Putting those concepts into practice by actually using them to achieve a goal is often the best way to understand them in detail. It also provides an opportunity to dig beneath the surface and get to know some of the pitfalls and finer points that hands-on experience brings.

The course project is your opportunity to get some real-world experience with these concepts. Each module has an associated **Project Exercise**, which will demonstrate your mastery of the core concepts introduced in that module. As you progress through the course, these exercises will build on each other to produce a final project that makes use of all the key concepts, tools and skills we'll cover.

Each exercise has a set of **Core Goals** that you must complete and submit, plus some **Stretch Goals** that go beyond the concepts covered by the reading material and provide an opportunity for you to delve deeper into some more advanced topics. You can choose to include these goals in the code you submit, but it is not a requirement for the course.

Let's get started!

Module 1

# Building a To-Do Web App

## Introduction

In this first exercise of the course, we're going to create one of the classic starter projects: **a To-Do app!**

We'll be developing this app throughout the course, with the ultimate goal of having a fully automated pipeline to build, test and deploy it to a cloud platform.

This exercise gives you a chance to practice your Python and become familiar with a popular web framework called Flask (https://palletsprojects.com/p/flask/).

If you get stuck working through the exercise steps, then there are lots of places to look for help:

- If you're stuck on how to use a particular tool or framework, don't forget to check their docs
- There are hints at the bottom of this document for common issues
- There are course-wide FAQs including exercise specific guidance for a wider range of potential tripping points - this might be worth bookmarking for future modules!
- Ask other people in your cohort - you should have a Slack or Teams space to do that and your peers will commonly have seen similar issues.
- Reach out to the tutors - we're more than happy to provide guidance!

### Exercise structure

Each project exercise will follow the same structure:

1. **Introduction:** what we'll be adding to the course project in this exercise.
2. **Setup:** any steps you need to follow first, in order to be able to complete the exercise.
3. **Exercise:** the core goals that you'll need to complete before submitting your work.
4. **Stretch goals:** optional additions you can make to the project that go beyond the core concepts for this module.
5. **Hints:** some pointers to helpful resources if you find yourself stuck.

## Setup: Checkout the starter project

Before you start, this exercise assumes you have already set up certain prerequisites. Check that you have done all the following before you continue.

- Installed Python on your machine.
  Instructions: python.org/downloads.

- Installed git on your local machine.
  Instructions: git (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git).

- Have (or have created) a GitHub account.
  Instructions: github.com/join.

### Step 1: Clone the starter project

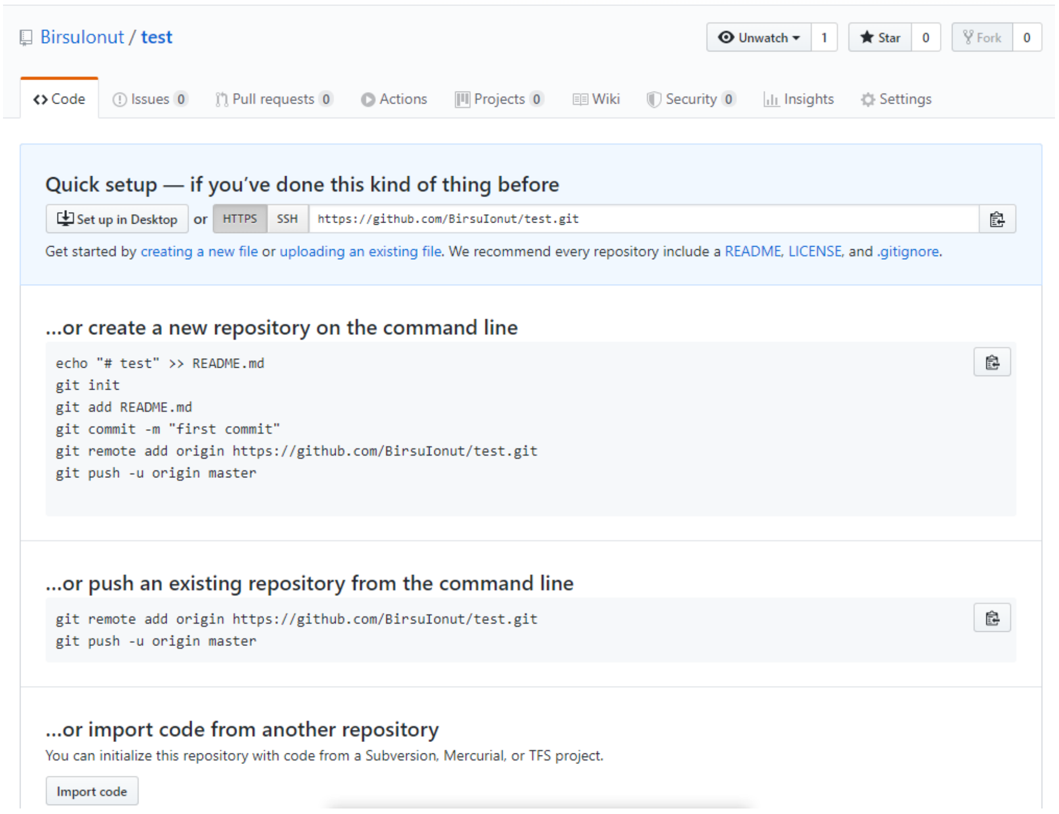
We've created a repository on GitHub containing the initial project files to get you up and running. Start by cloning this repo:

- Choose a suitable folder on your hard drive to contain the files for this course (we suggest somewhere in your user folder e.g. `C:\Users\{Your Username}` on Windows).
  - If using Windows, open that folder in File Explorer, right-click anywhere within it and select "Git Bash Here" from the menu that appears.
  - This will open a Git Bash terminal with that folder set as the current working directory.
- You can then run git clone to create a local copy of the repository:

```
$ git clone https://github.com/CorndelWithSoftwire/DevOps-Course-Starter.git
$ cd DevOps-Course-Starter
```

Next you'll want to push up your own copy of the repository up to GitHub:

- Create a new repository, giving it the name `DevOps-Course-Starter` and leaving everything else untouched. You should now see this page:



Quick setup — if you've done this kind of thing before

https://github.com/BirsuIonut/test.git

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# test" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/BirsuIonut/test.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/BirsuIonut/test.git
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

- Run `git remote remove origin` to unlink your repository from the original starter project (github.com/CorndelWithSoftwire/DevOps-Course-Starter)
- Follow the instructions under **"...or push an existing repository from the command line"**
  - The first command links your local repository to the newly created one under your account in GitHub
  - The second pushes your local repository up to the remote repository
- Finally you'll want to setup your tutors as collaborators on the repository so they can review your changes.
  - You can do this by selecting settings => manage access => invite a collaborator
  - Currently the reviewers are Stephen Shaw (`aeone`), Alex Jones (`Jonesey13`), Hugh Emerson (`hugh-emerson`), Jack Mead (`JackMead`) & Ben Ramchandani (`BenRamchandani`)

## Step 2: Installing and Running the Flask Web Application

We'll be using the Flask web framework and a couple of other packages in our app.

The codebase contains a `README.md` with instructions on how to setup the application and get it up and running.

After following the instructions you should see output similar to the following:

```
 * Serving Flask app "app" (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C
to quit)
 * Restarting with fsevents reloader
 * Debugger is active!
```

You should then be able to visit http://localhost:5000/ in your web browser and see the app in all its glory!

### Don't forget to commit!

Make sure you use **`git commit`** to save your work regularly.

It's good practice, and lets you revert back to a previous revision if you find you've broken everything horribly!

Try to make your commit messages concise and descriptive. Using source control tools effectively is just as important as the code you submit.

## Exercise: View and create todo items in the web app

In this exercise, you will need to:

- Make the app render the index page HTML template.
- Display the list of saved todo items on the index page.
- Create a route to add a new todo item to the list.

### Show the index page template

The starter project contains a template for the index page (`index.html` in the "templates" subdirectory). For more details about how to use templates, have a read through the official Flask tutorial (although this is quite technical; a more accessible tutorial can be found here).

The `render_template()` function allows you to render the specified template as HTML and return it as the response. Use it to replace the "Hello World!" response in app.py with the index page content.

### Display the list of saved todo items

We have provided a module (`session_items.py`) containing various functions to get, create and save todo items, which you can call directly from your view functions in `app.py`.

- This module uses a flask session cookie under the hood to keep track of the todos (but you don't actually need to know this to complete the exercise).

### CRUD

If you think about most actions users take within a typical application in the context of *resources* — i.e. the thing those actions are acting on — then a few common patterns tend to emerge.

Users can typically **create** resources (e.g. create a new order on a shopping web app, or add a new event in a calendar desktop app).

They can also view, or **read**, existing resources (view their order or see a calendar of all upcoming events). This could either be reading the details of a specific resource or viewing a list of all resources.

Often users can also edit, or **update**, existing resources (add items to their order; update the start time of an event).

Finally, they can **delete** an existing resource (cancel their order or remove an event).

These four actions — Create, Read, Update and Delete — are so common that they've come to be referred to as **CRUD** actions.

Note that most apps will usually have other, non-CRUD actions, too!

The main functions of interest are:

1. `get_items()` — returns the list of saved todo items.
2. `add_item(title)` — adds a new item with the specified title.

Modify the `index()` function to get the list of items and update the `index.html` template to display their titles as a list.

***Note**: the helper module contains code to pre-populate some todo items for you, so you won't start with an empty list! You will need to import those functions at the top of the file, e.g. with* `from todo_app.data.session_items import get_items`

## Create new todo items

We want to be able to add new items to our todo list. In order to support this functionality, we'll need to add a new route to the app.

By default, Flask treats each route as if it's returning a web page or some other data like an image. These routes are triggered when an HTTP GET request is made, e.g. by typing the address into your browser's address bar or clicking on a link.

In this case, however, we'd like to do the reverse; we'd like to send data to the Flask web server itself. The idiomatic way of doing this is to use an HTTP POST request.

We want to follow this convention, so add a new route to the app that accepts HTTP POST requests. You can use the `methods` argument of the `route()` decorator to specify which HTTP method(s) it will handle.

In order to capture the title for the new item, you'll also need to add a form to the `index.html` template. Use the URL of the route you just created as its "action" and use `method="post"` to submit a POST request.

Your form will need to contain an input field for the item title and a button element to submit the form data.

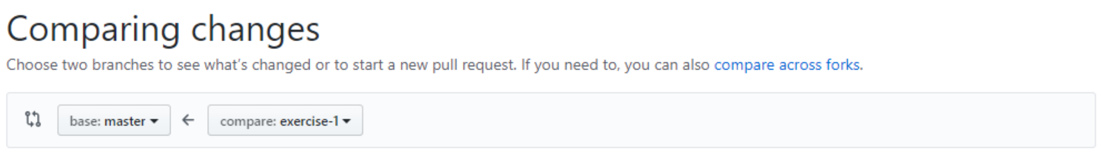
Update the function for your new route to retrieve the item title from the form data using `request.form.get('field_name')` — where `field_name` is the name of the input field in your form.

Finally, add the item to the list, and redirect the user back to the index page, using the `redirect` function from Flask. Returning a redirect means you avoid repetition of code, plus the browser will show the correct URL and can refresh without resubmitting the form.

## Submitting your work

We'll be using Pull Requests on GitHub to review exercise submissions during the course. To create a pull request once you're happy with your work:

- Make sure all your changes have been made on a new branch with the module number (e.g. `exercise-1`)
- Push that branch to your remote repository on GitHub, e.g. `git push origin exercise-1`
- Visit your repository on GitHub and create a pull request for the branch you just pushed. The pull request should merge the `exercise-1` branch into the `master` branch:


Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

Github pull request

  - Ensure that the `master` branch matches the original state of the repository (i.e. it is at the same commit as when you setup the repository)
    - If this is not the case then you'll want to reset this branch back to its original commit
- Please add your tutors as reviewers to the pull request (there should be option for this on the right hand side of the pull request page)
- Submit the URL for the pull request as your exercise solution on Aptem.

You can submit further changes to the pull request once it's created by committing those changes to your local branch (`exercise-1`) and then pushing them to the remote branch on GitHub. The pull request will update automatically.

## Submission checklist

A non exhaustive list of things to check when creating your Pull Request:

- Anyone could run the app following the README instructions
- You can create a new to-do item and see it immediately appear on the page
- You're raising the Pull Request against your own repository (e.g. `github.com/YourUserNameHere/My-Project-Name`, not `github.com/CorndelWithSoftwire/DevOps-Course-Starter`)
- Your repository is public, or you've added all the tutors as Contributors (see the end of Setup Step 1)

Once you've created the pull request and submitted it to Aptem we'll review it and give you feedback. The checklist above forms the basis of our marking criteria, so if you've done the above we will usually "Accept" the submission in Aptem. If not we'll make it clear what the problem is and why it is important.

The tutor reviewing your code will usually leave some extra comments even when accepting your submission. If you can it's always good to have a look at all the comments, but if you're pressed for time it should be clear whether these are optional or not - ask if you're not sure. If it is approved in Aptem you can always merge and include any follow up changes in your next submission.

If we mark your submission as "Referred" then you'll need to make the suggested changes, push them to the existing branch (which will update the pull request) and then resubmit in Aptem.

## Stretch Goals

- Add functionality to mark an item as completed
  - We've provided `get_item(id)` and `save_item(id)` functions as part of the `session_items.py` module which should help.
- Improve the styling of the app
  - We're using Bootstrap to style the pages; their documentation should provide some inspiration.
- Sort the item list by status ("Not Started", then "Completed")
  - Python's built-in `sorted()` function is a good place to start.
- Add functionality to remove an item
  - You will need to add a function to `session_items.py` to do this, as well as a route and HTML elements to trigger it.

### Stretch yourself

You'll notice that these stretch goals are more ambiguous and open-ended than the core goals for the exercise.

These are an opportunity for you to apply your creativity and decide how best to implement new features in the app.

You'll also probably need to do some additional research to find out how to achieve these goals, so break out a search engine and get exploring!

## Hints

- We've provided a number of helper functions to retrieve and save todo items. You can use these functions directly in your routes in `app.py`, for example: `items = session.get_items()`
- To see how the functions work, take a look at `session_items.py`
- The Flask tutorial is a good resource for info and examples if you get stuck, particularly the sections on templates and views.
- If you are unfamiliar with HTML, it might be worth reading this tutorial about forms.